TÉCNICO LISBOA

Instituto Superior Técnico

Parallel and Heterogeneous Computing Systems

1º Semester 2021/2022

1º Report
Parallel Fractal Generation Using Threads

Group 7

Nome	Número
Pedro Silva	87103
André Pereira	90016
Filipe Santos	90068

Day of the week: Monday, from 11:30 AM to 2:30 PM.

Professor: Aleksandar Ilic

1 Problem 1

1.1 Performance results

Table 1: Performance Results for the original program parallelized

Number of Threads	$Serial_{milliseconds}$	$Threads_{milliseconds}$	Speedup	Thread Usage
2	393,89	201,019	1,959466518	97,97%
3		239,454	1,644950596	54,8%
4		162,843	2,418832864	60,47%
5		158,083	2,491665771	49,83%
6		126,499	3,113779556	51,89%
7		119,152	3,305777494	47,22%
8		101,905	3,86526667	48,31%
16		68,822	5,723315219	35,77%

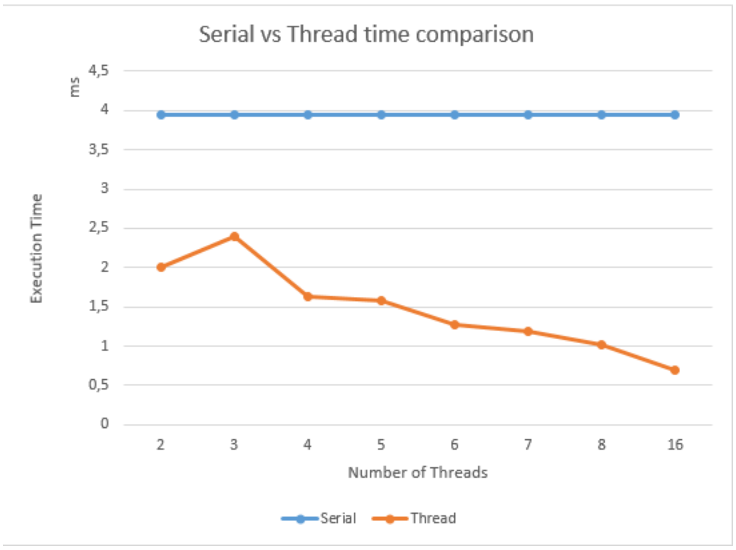


Figure 1: Comparison between best threaded algorithm results with serial implementation

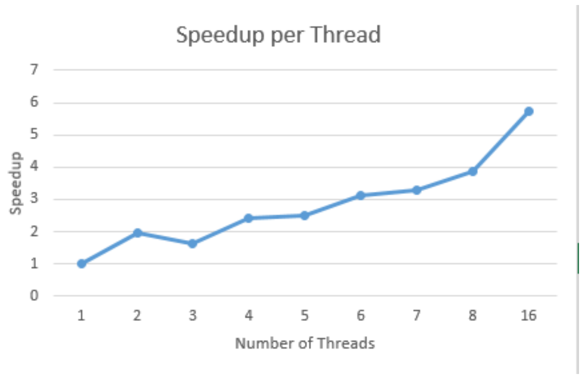


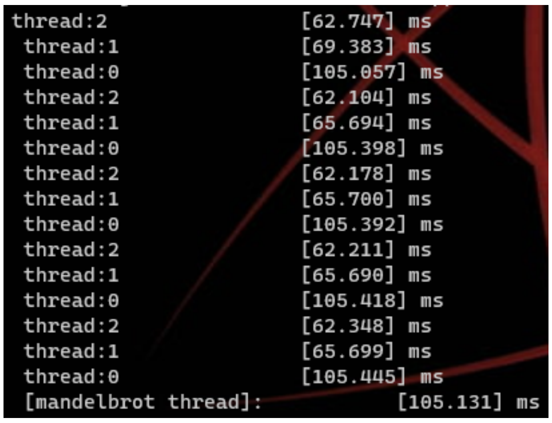
Figure 2: Speedup achieved by parallelizing the original algorithm

Ideally, we expect a speedup equal to the number of threads if the program was fully parallelizable.Theoretically, the program will always have overheads associated with threads creation and communication, as well as workload balancing, making it impossible to achieve a speedup equal to the number of threads. Looking at the results, we can see that the speedup was not linear with the number of threads used. In this problem, the major problem that stops us from achieving better speedups is the workload balance.

1.2 Workload Balance

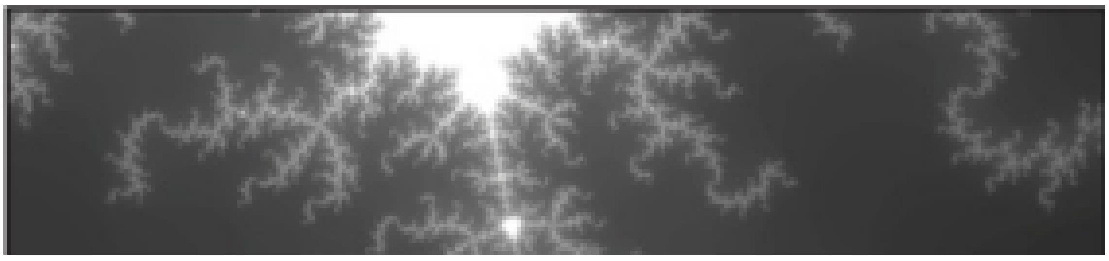
By dividing the image in horizontal sections to be processed by different threads, some threads will need to have more intensive computations done constantly to calculate the fractal (the set of complex numbers).

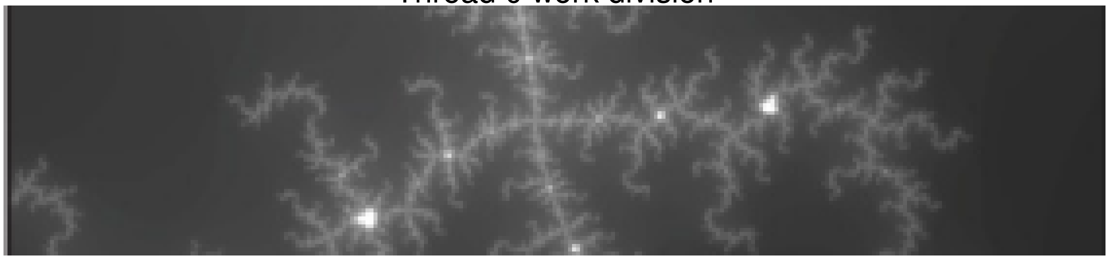
For example, if we look at the view 2 and the corresponding thread work times:



```
thread:2                [62.747] ms
 thread:1               [69.383] ms
 thread:0               [105.057] ms
 thread:2               [62.104] ms
 thread:1               [65.694] ms
 thread:0               [105.398] ms
 thread:2               [62.178] ms
 thread:1               [65.700] ms
 thread:0               [105.392] ms
 thread:2               [62.211] ms
 thread:1               [65.690] ms
 thread:0               [105.418] ms
 thread:2               [62.348] ms
 thread:1               [65.699] ms
 thread:0               [105.445] ms
 [mandelbrot thread]:           [105.131] ms
```

Figure 3: Time each thread spent working


Thread 0 work division


Thread 1 work division

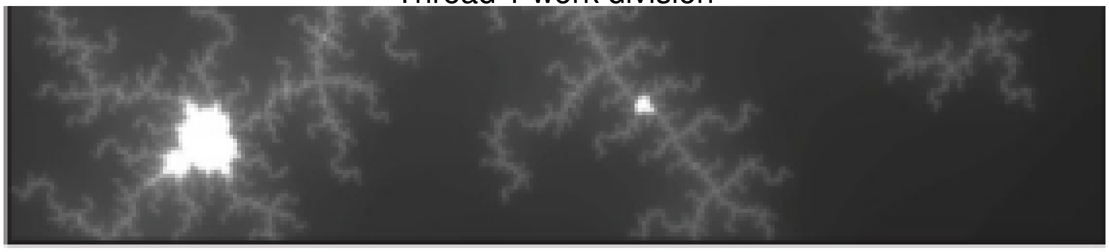
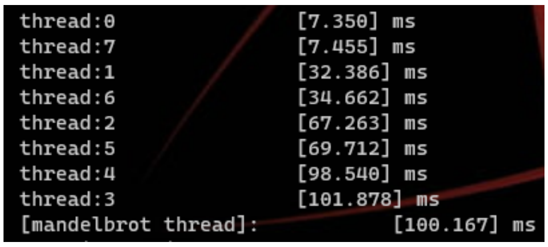

Thread 2 work division

Figure 4: Division of view 2 figure by each thread

We can see that thread 0 takes much longer to complete than thread 1 and 2. This is because thread 0 has more intense computation to perform right around the middle of the photo, on the large white zone, and has a lot of branches branching out from it. Even thought all 3 threads have the same amount of data given to them, thread 0 has to spend longer computing the complex numbers from it´s workload than the other 2 threads, negatively impacting the overall performance of the algorithm. The brightness of each pixel is proportional to the computational cost, and thread 0 has a lot of white pixels when compared to the other two threads.

These measurements confirm that the algorithm cannot properly take full advantage of having more threads to achieve faster times and higher speedups, since the overall time of the algorithm is dictated by the slowest working thread, as we can see in view 1:



```
thread:0                  [7.350] ms
thread:7                  [7.455] ms
thread:1                  [32.386] ms
thread:6                  [34.662] ms
thread:2                  [67.263] ms
thread:5                  [69.712] ms
thread:4                  [98.540] ms
thread:3                  [101.878] ms
[mandelbrot thread]:              [100.167] ms
```

Figure 5: Time each thread spent working on view 1

Here, we have a couple of threads that need mere miliseconds to perform the computation of the complex numbers, while others take more than 10 times longer to perform said calcultions. This means that various threads are just left waiting with no work to perform, rendering them useless. If all 8 threads (in this case) were working simultaneously, the speedup achieved would be much closer to 8, instead of the obtained 3,86.

1.3 Modification of the work mapping

To fix this uneven work balance, we decided to split each line interleaved with each thread. So, for example, if we have 3 threads, we give the first line of the matrix to thread 0, the second line to thread 1, the third line to thread 2, the fourth line to thread 0, the fifth line to thread 1, so on and so forth...

By doing this, we can make sure that all threads have a simillar intensity of computation in their datasets. Since the image works like a gradient (pixels that are in the vicinity of others will have close colors), then there will not be a thread that takes care of a single section of a image that might take a large area of white pixels. These sections will be evenly divided between all threads, achieving better work load balances and, therefore, better performance results, as we can see in table 2:

Table 2: Performance Results for the new workload division program

Number of Threads	$Serial_{miliseconds}$	$Threads_{miliseconds}$	SpeedUp	Thread Usage
8	393,89	54,772	7,18	89,75%
16		58,25	6,76	42,25%

1.4 16 Threads

By running the original algorithm parallelized with 16 threads, we achieve a speedup greater than having only 8 threads, but the ratio of threads created by the speedup achieved is severely lower than what was expected for the 16 threads. If we look at the Thread Usage column of table 1, we can detect a pattern: the speedup is around 50% of the number of threads used. But, when looking at 16 threads, the ratio that we observed plumets to, roughly, 36%. This happens because the machine were these tests were ran was the cuda2 machine, that has available 4 physical cores, and each core supports two hyper-threads. Therefore, this machine has 8 logical cores. When using less than 9 threads, it is easy to assign a thread to each logical core, performing all the work there. When creating and assigning more that 8 threads, we have a problem: we don´t have a dedicated logical core to each of the threads created, so the computer will need to schedule when and which thread will perform some amount of work in a core. This scheduling takes up resources that, ultimately, could be used to achieve better speedups. So, in theory, the execution time for 16 threads should have been worse than the 8 threads, not just the thread usage ratio. But, in the original program, the workload is not evenly balanced, so, by dividing more the image, the workload is much better balanced between each of the 16 threads when compared to the 8, so the time it took to ran the program was actually faster, even with the added overheads of having more threads.

Nevertheless, if we now run the same test with the new work mapping, we can finally achieve the results we were hoping for. By using only 8 threads, we achieve a faster time than using 16 threads, because the workload balance is much better than the previous mapping, and the overheads of having more threads finally weights more in the execution time. This is seen in table 2.

2 Problem 2

In this problem, a program was vectorized using “fake” SIMD Intrinsics just to check the performance of vectorized code. The performance of each SIMD width can be checked on 3:

Table 3: Performance Results for SIMD width

Vector length	Total Vector Instructions	Vector Utilization (%)	Execution time[cycles]
2	181	78,18	264
4	119	72,48	181
8	74	63,18	112
16	40	63,44	60

As expected, as the vector length increases, the execution time decreases since more values of the vector are being computed in parallel. It doesn’t have a linear behaviour because some elements take longer than others, which means, for a certain vector, the computation loop repeats itself depending on the longer computation element. That is why, vectors with half the length don’t necessarily have double the instructions. This is also the reason why vector utilization increases as the vector length decreases. With shorter vector lengths, there are less elements waiting for the longer computation element. This means that more elements are active at all times. The only time the vector length doesn't increase is from SIMD 16 to SIMD 8. This is due to the three instructions that are out of the loop that have 100% vector utilization, which in the SIMD 16 has a heavier impact, since this one has less number of vector instructions overall.

3 Problem 3

3.1 ISPC Basics

Table 4: Performance Results for each view

View	$Serial_{miliseconds}$	$ISPC_{miliseconds}$	SpeedUp
View 1	197.016	38.164	5.16
View 2	115.904	26.683	4.34

When we ran the program, we ideally expected a speedup of 8 for both views, since we executed the program in a single core using 8 execution lanes. Instead, as seen in table 3, we only achieved speedups between 4 and 5. This comes from the fact that the workload balancing also comes in effect during the different instances running. If we look at figure 6 and 7:

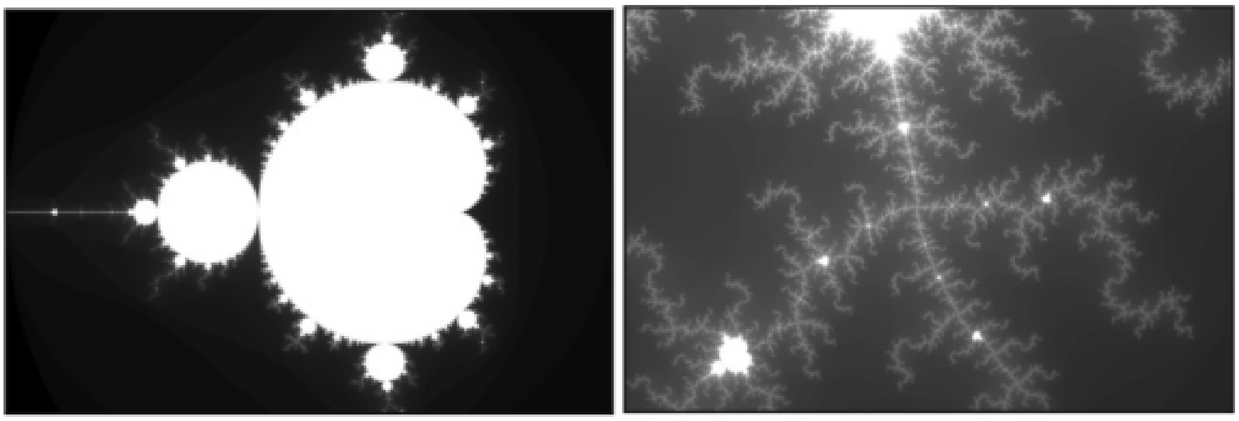

(a) Fractal of view 1 (b) Fractal of view 2

Figure 6: Both Fractals side by side

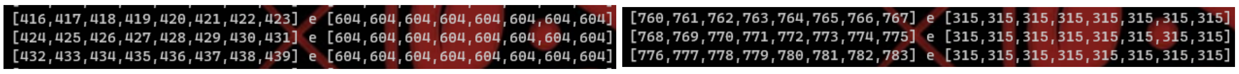

```
[416,417,418,419,420,421,422,423] e [604,604,604,604,604,604,604,604]
[424,425,426,427,428,429,430,431] e [604,604,604,604,604,604,604,604]
[432,433,434,435,436,437,438,439] e [604,604,604,604,604,604,604,604]
```
```
[760,761,762,763,764,765,766,767] e [315,315,315,315,315,315,315,315]
[768,769,770,771,772,773,774,775] e [315,315,315,315,315,315,315,315]
[776,777,778,779,780,781,782,783] e [315,315,315,315,315,315,315,315]
```
(a) Division of i and j in the foreach loop: view 1 (b) Division of i and j in the foreach loop: view 2

Figure 7: Workload division between ISPC instances

We can see that view 1 has a lot of regions where horizontal neighbour pixels have close pixel intensities, while view 2 has the opposite, having lots of regions where horizontal neighbour pixels have different pixel intensities (because of all the small branches in the fractal). For each loop iteration, the foreach construct distributes 8 pixels in a row, horizontally, through the instances. In view 1, these 8 pixels will have a computational cost close to each other, while in view 2 this does not occur (in view 1, when we take a chunk of 8 pixels, they will most likely be a set of 8 black or 8 white pixels, while in view 2 the chunks will more frequently chunks that are a mixture of grey and white pixels). Therefore, in view 1, the time each instance takes to compute it´s pixels is around the same time that it takes to all of the instances to compute their pixel, but in the other view, the time it takes a instance to compute it´s pixel is not the same as all the other instances. This means that the instances will remain idle while the instance with the heaviest computational cost will be calculating it´s pixel. A new chunk will only be given when the last instance finish it´s work. But both views will have chunk attribution with uneven workload distribution between instances. Therefore, the speedup can never be 8 for any of them, and view 1 has a faster speedup than view 2 because of what was explained previously.

3.2 ISPC Tasks

The following results were obtained by doing an average of 10 runs:

Table 5: Performance Results for creating tasks for view 1

Number of Tasks	$Serial_{milliseconds}$	$ISPC_{milliseconds}$	Speedup
1		38.942	5.06
2		19.601	10.05
200	197.016	5.619	35.057
400		5.604	35.157
800		5.654	34.841

In general, one should launch many more tasks than there are processors in the system to ensure good load-balancing, but not so many that the overhead of scheduling and running tasks dominates the computation. Thus, we did some testing. We divided the job of computing the fractals into small tasks: either a task is to compute a row (800 tasks), two rows (400 tasks) or 4 rows of pixels (200 tasks). By analyzing the speedups obtained, we can concluded that the best workpool of tasks that should be used is the 400 one. If we have a workpool of the 200 tasks, there is still a better speedup that we can achieve, but if we do the 800 tasks, since each job of a task is so small, there is a bit of overhead that dominates the computation time more than the 400 tasks pool.

4 Problem 4

The program computes the square root of 20 million random numbers between 0 and 3, by using Newton's method. It was also given a graph showing iterations until convergence for this algorithm depending on input value. It should also be noted that this implementation has 1 as it's initial guess, and, as one could predict, it would therefore have the fastest convergence.

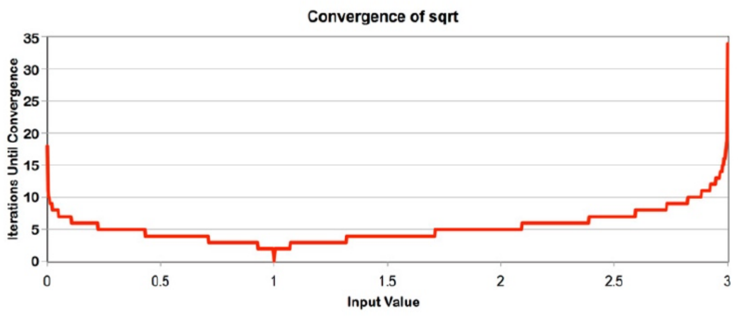


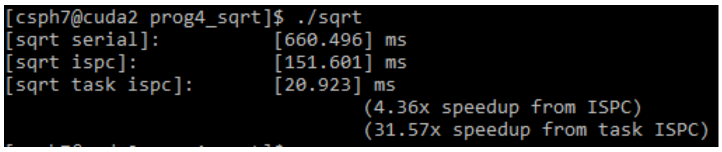
Figure 8: Graph of convergence for this implementation

After running the sqrt program with the random vector, the speedups were the following (also shown in figure 9).

Table 6: Performance Results for random input case

	$Serial_{milliseconds}$	$ISPC_{milliseconds}$	Speedup
SIMD parallelization Multi-core parallelization	660.496	151.601 20.923	4.36 31.57

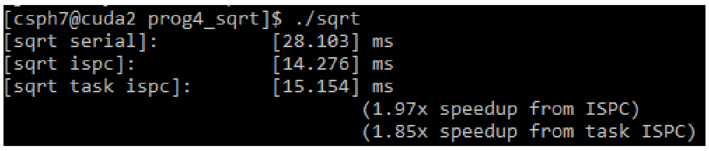
Although still far away from the optimal 8x for regular ISPC and 64x for the task implementation, they're already quite significant.



```
[csph7@cuda2 prog4_sqrt]$ ./sqrt
[sqrt serial]:          [660.496] ms
[sqrt ispc]:            [151.601] ms
[sqrt task ispc]:       [20.923] ms
                                (4.36x speedup from ISPC)
                                (31.57x speedup from task ISPC)
```

Figure 9: Speedup achieved for random numbers between 0 and 3

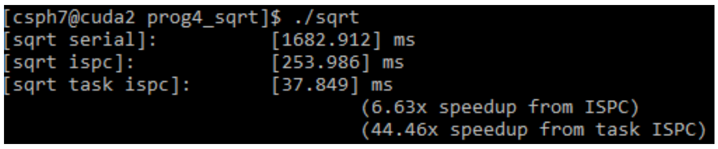
Given the previous statement, we also checked the speedup for an array full of ones, expecting a lower speedup for this case. It was verified in testing, and speedups lower than 2x were recorded. The reason this value is so low is that for an input value of 1, the square root function can be summarized to a for loop with 7 lines of code, and therefore a respectively lower instruction count, which leads to losing the benefits of being parallelized. One can even verify that, since the iterational computational load is so low, further division in deploying tasks even warrants slower times because of its overhead trade off.



```
[csph7@cuda2 prog4_sqrt]$ ./sqrt
[sqrt serial]:          [28.103] ms
[sqrt ispc]:            [14.276] ms
[sqrt task ispc]:       [15.154] ms
                                (1.97x speedup from ISPC)
                                (1.85x speedup from task ISPC)
```

Figure 10: Speedup achieved with every entry as 1

Taking into consideration the nature of Newtons method, it is expectable that the higher the distance to the initial guess, the slower the convergence would be, as one can somewhat verify in the graph. This does not happen linearly across a set of positive numbers due to how the square root function works, but it is verifiable for the interval $\mathbb{R}$ =]0,3[. The program was ran with the array populated with entries of 3, and greater speedups, compared to the random input case, were confirmed.



```
[csph7@cuda2 prog4_sqrt]$ ./sqrt
[sqrt serial]:          [1682.912] ms
[sqrt ispc]:            [253.986] ms
[sqrt task ispc]:       [37.849] ms
                                (6.63x speedup from ISPC)
                                (44.46x speedup from task ISPC)
```

Figure 11: Speedup achieved with every entry as 3

Looking at the speedup values for this last case, it is assumed that the amount of iterations until convergence isn't high enough to attain optimal speedups of 8x and 64x or approximate values. It was already verifiable in the second case analyzed that the associated costs can overshadow performance improvements. In this one, the increase in iteration count, which translates to more than 10x the load, still ends up not being enough to compensate fully for the other costs.

5 Problem 5

In this case, with tasks, the performance actually gets slightly worse than with no tasks. This decrease in performance is due to the overhead of scheduling and running tasks that, in this case, is more notable than the improvement by dividing the work in tasks. To prove this, a change was made in the program, to compute the same values various times to increase computation load. With this change, the performance with tasks presents a speedup in relation to the program with no tasks.

In this program it's not very good to use ISPC because there is very little computation, and the overhead associated with the creation of tasks proves to be prevalent over the improvement on work balance.